Phrase Matching in XML

Sihem Amer-Yahia, Mary Fernández, Divesh Srivastava
AT&T Labs–Research
{sihem, mff, divesh}@research.att.com

Yu Xu
UC, San Diego
yxu@cs.ucsd.edu

Abstract

Phrase matching is a common IR technique to search text and identify relevant documents in a document collection. Phrase matching in XML presents new challenges as text may be interleaved with arbitrary markup, thwarting search techniques that require strict contiguity or close proximity of keywords. We present a technique for phrase matching in XML that permits dynamic specification of both the phrase to be matched and the markup to be ignored. We develop an effective algorithm for our technique that utilizes inverted indices on phrase words and XML tags. We describe experimental results comparing our algorithm to an indexed-nested loop algorithm that illustrate our algorithm's efficiency.

1 Introduction

XML, and its ancestor SGML, were originally developed by the document processing community for adding both structural and semantic markup to texts. You are probably familiar with Shakespeare's plays, which have been augmented to include markup describing scenes, speeches, and speakers [9]. Classical literature abounds in commentaries added by literary critics (e.g., the Talmud contains commentaries on Biblical text). XML permits such commentaries to be easily identified via user-defined annotations. As a more recent example, the XML documents published by the Library Of Congress (LOC) [17] contain the large texts of legislative bills; in these texts, the names of the sponsors of a bill and the committees to which a bill is referred are identified in the body of the bill with markup. XML can also be used to represent the output of natural-language processing systems; such output labels the grammatical structure of natural language text, for example, with subjects and verbs, and noun and adjective phrases [18].

In the absence of markup, *phrase matching* is a common technique to search text and identify relevant documents. Phrase matching typically requires that words in a phrase be contiguous or in close proximity. For example, searching for the phrase "To be, or not to be" would return very different results than searching for the same set of words as individual keywords. Most information retrieval (IR) systems support phrase matching on text and on HTML documents, ignoring HTML tags to match phrases.

In extending phrase matching to XML, which permits arbitrary user-defined markup, it should be possible to specify the individual tags and the complete annotations (i.e., elements and their content) to ignore. For example, to match the phrase "Mr. English introduced this bill" in this XML document fragment:

```
<sponsor>Mr. English</sponsor><footnote>For
himself and<co-sponsor>Mr. Coyne</co-sponsor>.
</footnote> introduced this bill, which was
referred to the <committee-name>Committee
on Financial Services</committee-name>
```

it is necessary to ignore the `<sponsor>` end tag, and the entire `<footnote>` annotation. Specifying that the `<co-sponsor>` tag should be ignored does not change the answer, but not specifying the `<sponsor>` tag does.

Such customized phrase matching is not supported by existing IR systems, but future systems could provide it in various ways. A naive-user interface might specify a priori the tags and annotations to ignore given knowledge about the application domain and the schemas for input documents. An interface to the LOC archive, for example, might automatically ignore the `co-sponsor` and `sponsor` tags and the `footnote` annotations. An expert-user interface might permit the user to specify the ignored markup, providing him with more control over phrase matching. Finally, such customized phrase matching could be easily integrated into XML query languages, such as XQuery [7], providing applications with all of XQuery's functionality in conjunction with phrase matching.

Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003

We present an approach to phrase matching, tailored to XML documents, that permits specification of the phrase to be matched and the tags and annotations to be ignored. Our approach is appropriate for systems that specify ignored markup a priori or dynamically. In particular, our contributions include:

- An efficient algorithm, PIX, for phrase matching in XML documents that utilizes inverted indices on phrase words and tags. PIX supports phrase matching with strict contiguity of phrase words and also within a proximity of k words. PIX is optimal among algorithms that scan the complete inverted lists of query terms and is implemented as an external, user-defined XQuery function in the Galax [14] implementation of XQuery.
- An experimental evaluation comparing the PIX algorithm with an indexed nested loop (INL) algorithm that utilizes the same set of indices as PIX. Results show that PIX is substantially more efficient than the INL algorithm on XML documents that contain nested matches or many tags and annotations to ignore, and on queries whose phrases contain many words.

We begin in Section 2 with an example that illustrates the requirements for phrase matching in XML documents. Section 3 describes research relevant to phrase matching from the domains of information retrieval and query processing. Section 4 presents the INL and PIX algorithms. The results of experiments comparing exact phrase matching using PIX and the INL algorithm appear in Section 5.

2 Motivating Example

To illustrate phrase matching in structured documents, we present a fragment of the XML for Shakespeare's play Hamlet in Figure 1.[1] Although an example from literature may seem unusual, it has many characteristics of XML in document-processing applications, including markup that tags semantically significant text and annotations that may contain corrections, clarifications, or instructions.

This document contains large bodies of annotated text, which is referred to as *mixed content* in XML. In addition, the document may contain structured *regular data*, such as the play's title, its playwright and year of publication. Regular data is typically strictly typed (e.g., an `year` element must contain a valid date). Constraints on regular data (e.g., element types) and on mixed content (e.g., permissible markup) are usually expressed in some XML schema language.

Although regular data and mixed content look similar, they differ in practice, because they are queried in different ways. Queries on regular data usually include exact (in)equality predicates over small typed values, whereas queries on mixed content include keyword matches and exact or approximate phrase matches over large fragments of text. XQuery currently provides many constructs for querying regular data, and there is growing interest in extending XQuery for effectively querying mixed content [2].

To illustrate how matching phrases interacts with XML markup, we consider the phrases in Table 1. The table also contains the matches of these phrases against the text in Figure 1. Suppose we first match these phrases literally against the XML document[2]. Phrase P1 is matched once on line 28. None of the other phrases match, because of intervening markup. Phrase P1 is not matched on lines 26 and 31, because of an intervening `<COMMENT>` annotation on lines 27–30; similarly, phrase P2 is not matched on lines 10 and 12, because of an intervening stage direction (within `<STAGEDIR>` and `</STAGEDIR>` on line 11) and because of additional line boundary markup (`</LINE>` on line 10 and `<LINE>` on line 12).

IR search engines typically ignore markup in HTML documents when matching phrases. If we ignore all markup in our example XML document, phrase P4 now matches on line 19 since `<PP>` is ignored, but phrase P1 still does not match on lines 26 and 31 nor does phrase P2 match on lines 10 and 12, because the content of the annotation and stage direction elements intervene. Also, phrase P5 is matched spuriously on lines 36-39, across multiple speeches, which is undesirable. These examples motivate differentiating between two types of markup when matching phrases: individual *tags* and complete *annotations*.

Specifying the tags and annotations to ignore yields more matches. If we ignore the annotation in `<COMMENT>...</COMMENT>`, then phrase P1 matches on lines 26 and 31. If we ignore the annotation in the parenthetical phrase `<PP>...</PP>` and the individual `LINE` tags, then phrase P3 matches (but phrase P4 does not), but if we just ignore the individual `PP` tags, then phrase P4 matches (but phrase P3 does not). The choice of whether to ignore `<PP>` as an annotation, or as an individual tag, thus depends on the specific query. Similarly, if we ignore the entire `STAGEDIR` annotation and the individual `LINE` tags, phrase P2 matches. We also note that there may be multiple matches and that they may be nested within annotations. The match of phrase P1 on line 28 in the `COMMENT` annotation, for example, is nested within the match in lines 26 and 31 in the original text.

So far, our examples have required exact matches in which every word in the phrase must occur contiguously in the text (after ignoring specified tags and annotations), but approximate phrase matching is also common in practice. One kind of approximation is to

[1] This fragment is from Jon Bosak's collection of Shakespeare's plays [9] in XML, annotated with additional markup and commentary.

[2] As in IR-style search, punctuation is usually ignored.

```
01: <PLAY>
02: <TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE><YEAR>1601</YEAR>
03: ...
04: <SPEECH>
05:   <SPEAKER>HORATIO</SPEAKER>
06:   ...
07:   <LINE>Speak to me:</LINE>
08:   <LINE>If there be any good thing to be done,</LINE>
09:   <LINE>That may to thee do ease and grace to me,</LINE>
10:   <LINE>Speak to me:</LINE>
11:   <STAGEDIR>Cock crows</STAGEDIR>
12:   <LINE>If thou art privy to thy country's fate,</LINE>
13:   ...
14: </SPEECH>
15: ...
16: <SPEECH>
17:   <SPEAKER>KING CLAUDIUS</SPEAKER>
18:   ...
19:   <LINE>The harlot's cheek <PP>beautied with plastering art</PP></LINE>
20:   <LINE>Is not more ugly to the thing that helps it</LINE>
21:   ...
22: </SPEECH>
23: ...
24: <SPEECH>
25:   <SPEAKER>HAMLET</SPEAKER>
26:   <LINE>To be, or not to be:
27:     <COMMENT>
28:     The line <QUOTE>To be, or not to be: that is the question</QUOTE> is one of the
29:     most quoted phrases in the English language.
30:     </COMMENT>
31:   that is the question:</LINE>
32:   <LINE>Whether 'tis nobler in the mind to suffer</LINE>
33:   <LINE>The slings and arrows of outrageous fortune,</LINE>
34:   ...
35:   <LINE>The fair Ophelia! Nymph, in thy orisons</LINE>
36:   <LINE>Be all my sins remember'd</LINE>
37: </SPEECH>
38: <SPEECH>
39:   <SPEAKER>OPHELIA</SPEAKER>
40:   ...
41: </SPEECH>
42: </PLAY>
```

Figure 1: Example XML document fragment

Id	Phrase	Matches on lines
P1	To be or not to be that is the question	(26,31) and 28
P2	Speak to me if thou art privy	(10,12)
P3	The harlot's cheek is not more ugly	(19,20)
P4	The harlot's cheek beautied with plastering art	19
P5	remember'd Ophelia	None

Table 1: Example phrases to match

match phrase words within some "proximity" of other words. For example, the phrase "The harlot's cheek is ugly" matches "The harlot's cheek is not more ugly" within two intervening words. Many other approximations are possible, for example, matching the root of a word by applying a "stemmer" (a language-specific algorithm that determines the morphological root of an inflected or derived word form) or matching words that are semantically equivalent or similar to words in the phrase. Word approximation techniques are orthogonal to and, therefore, can be combined with techniques that ignore markup. Proximity queries interact with structure and deserve more attention.

Whether an IR system provides exact or approximate matching, matches are typically reported in a ranked order. A common ranking metric is the TF-IDF (term frequency/inverse document frequency) relevance measure, in which a higher weight is assigned to terms that occur frequently within one document and infrequently in the corpus of documents. The structure in XML documents adds another dimension to ranking, and a ranking function may also incorporate the properties of ignored tags and annotations.

Customized phrase matching in XML could be realized, for example, by an infix operator fully integrated into a query language such as XQuery, by a XQuery user-defined function, or by an external function (i.e., a function not defined in the query language itself). We have implemented PIX as an external function that can be called from the Galax implementation of XQuery.

This motivating example illustrates that phrase matching in XML documents is not a trivial problem. Solutions to this problem should permit dynamic (i.e., at query time) specification of ignored tags and annotations; handle multiple and nested matches; permit specification of arbitrary document fragments as the search context; and support approximate matching. The PIX algorithm meets these requirements: it permits specification of the phrase to be matched either exactly or within a word proximity, the document contexts in which to restrict the phrase match, and the tags and annotations that should be ignored. The algorithm can also rank the results during matching.

3 Related Work

XQuery 1.0 provides many constructs for querying regular data, such as path expressions for navigating document structure and predicate expressions for conditionally selecting fragments of documents. There is growing interest in extending XQuery 1.0 with full-text operators for effectively querying mixed content [2].

Information retrieval techniques, including exact phrase match and the approximation methods noted earlier, are described in classical IR texts [5, 19]. These techniques typically treat documents as "bags of terms" and employ inverted indices on terms in documents for fast lookup of specific terms. They do not address the problem of phrase matching within structured documents.

More recent techniques combine IR keyword matching with queries over document structure [4, 5, 11, 12, 13], often implemented as extensions to XPath or XQuery. An example that combines phrase matching with document structure might be: "for each text node that contains the phrase 'more ugly', return the node and its preceding and following sibling nodes". Such queries can be easily expressed in XPath, but require that all words in a phrase occur within one text node (i.e., words cannot be separated by intervening tags or annotations). XXL [20] and XIRQL [13] are examples of systems that focus on using word approximation techniques in combining IR search with XML structure. XQuery-IR [10] is an extension of XQuery that supports restriction of phrase matching to document fragments (as in PIX) and ranks document fragments by applying TF-IDF weights dynamically to document fragments. In the XKeyword system [16], words may be matched anywhere in a document and ranking is based on a document graph distance between the matched words. The XRank [15] system also supports keyword search and ranks results by combining the specificity of a match (i.e., all words in one element or within descendants of an element), the distance between keywords, and any intervening markup. A key difference with these previous systems is that PIX supports customized phrase matching, permitting the specification of which markup to ignore.

4 Algorithms

We now define two algorithms for phrase matching in XML documents, a simple indexed nested loop (INL) algorithm and the stack-based merge algorithm used in PIX. Both algorithms process document contexts in document order, keeping track of nesting of document contexts, potential matches, and ignored markup to minimize redundant traversals. We begin by defining the algorithms' required input, their expected output, and the inverted indices on words and tags that they use. The input to each algorithm is:

- Set of context tags $C = c_1, ..., c_m$
- Set of ignored tags $T = t_1, ..., t_k$
- Set of tags of ignored annotations $A = a_1, ...a_\ell$
- List of phrase words, in order $W = [w_1, ..., w_q]$

The set C contains the tags of nodes in which to restrict phrase matching. (In our example, `SPEECH` is a context tag). The set T contains the individual tags to ignore, the set A contains the tags of complete annotations to ignore within phrase matches, and list W contains the phrase words.

Before query processing, each element and text node in an input document is assigned a (*start*, *end*) interval, as in [1]. For our purposes, each text node contains one word, so we abbreviate a text interval (i, i) as i. Intervals permit fast checking of the descendant and following-sibling relationships. For example, if node n has interval (s, e) then any node n' with interval (s_i, e_i) such that $s < s_i$ and $e_i < e$ is a descendant of n; if $s_i = e + 1$, then n' is the first sibling node following n, i.e., n and n' are contiguous in the document. Figure 2 contains a fragment of our example document labeled with intervals.

The output of each algorithm is a set of (*context interval* i_c, *witness set* $\{m\}$) pairs. The interval i_c denotes an occurrence of a node whose tag is in C. Each of its witnesses is output, where a witness m is an ordered list of intervals $[i_1, ..., i_v]$ such that:

```
<SPEECH$_{(1,44)}$>
  <SPEAKER$_{(2,4)}$>HAMLET$_{3}$</SPEAKER>
  <LINE$_{(5,43)}$>To$_{6}$ be$_{7}$, or$_{8}$ not$_{9}$ to$_{10}$ be$_{11}$:
    <COMMENT$_{(12,38)}$>
    The$_{13}$ line$_{14}$ <QUOTE$_{(15,26)}$>To$_{16}$ be$_{17}$, or$_{18}$ not$_{19}$ to$_{20}$ be$_{21}$: that$_{22}$ is$_{23}$ the$_{24}$ question$_{25}$</QUOTE>
     is$_{27}$ one$_{28}$ of$_{29}$ the$_{30}$ most$_{31}$ quoted$_{32}$ phrases$_{33}$ in$_{34}$ the$_{35}$ English$_{36}$ language$_{37}$.
    </COMMENT>
  that$_{39}$ is$_{40}$ the$_{41}$ question$_{42}$:</LINE>
</SPEECH>
```

Figure 2: Example document fragment labeled with intervals

1. i_1 denotes an occurrence of word w_1, and i_v denotes an occurrence of word $w_q, v \geq q$,
2. i_c.start $< i_1$.start and i_v.end $< i_c$.end,
3. for each $1 \leq j < v, i_j$.end $= i_{j+1}$.start $- 1$,
4. there exists m', a subsequence of m of length q, such that the kth interval in m' denotes w_k, and
5. each interval in the remainder subsequence $m \setminus m'$ denotes an occurrence of ignored markup.

The first constraint guarantees that the first (last) interval in the witness denotes the first (last) phrase word. The second constraint guarantees the witness is contained within the given context. The third guarantees that the words and ignored markup in the witness are contiguous. The fourth constraint guarantees that all the words in the phrase occur in order. The last constraint guarantees that the remaining intervals in the witness denote ignored markup. Figure 3(a) contains the output of matching phrase P1 within a SPEECH context ignoring LINE tags and COMMENT annotations. It contains one result, which contains two matching witnesses.

Recall that both algorithms are dynamic, i.e., the phrases to match and the tags and annotations to ignore are not known until query time. Therefore, one inverted index is built off-line, in one pass, for *every* tag and word in the input document. Each index is a list of intervals sorted by start position and may be accessed sequentially. Each index is also a partial function from a start position to an interval, that is, given a start position i and index L, probe(L,i) returns the interval (i, j) if it exists in L. The partial function is implemented by a B-Tree over the sorted interval list. Figure 3(b) contains some of the indices for our example document fragment. Evaluating probe($\mathrm{L}_{"to"}$, 10) returns (10,10), and probe($\mathrm{L}_{"be"}$, 13) returns nothing.

At query time, the relevant indices are:

L_C**:** Index of all intervals of context tags in C.

L_{w_j}**:** Index of all intervals of word w_j.

L_{t_j}**:** Index of all intervals of ignored tag t_j.

L_{a_j}**:** Index of all intervals of ignored annotation a_j.

We also *conceptually* construct two more indices, which are never materialized:

LE_{t_j}**:** Index $\cup_j \{(s, s), (e, e)$ such that $(s, e) \in L_{t_j}\}$ sorted by first component.

L_M**:** Index of all ignored markup $\cup((\cup_j L_{a_j}), (\cup_j LE_{t_j}))$ sorted by first component.

For each ignored-tag interval in L_{t_j}, the index LE_{t_j} contains one interval for the start position and one for the end position; this allows the algorithms to skip over individual tags, but not their content. The index L_M is an interval list over all ignored markup in L_{a_j} and LE_{t_j}. Neither index is materialized, but is implemented using priority queues over the indexed lists L_{t_j} and L_{a_j}. Both algorithms use L_C, L_M, and L_{w_j}.

4.1 Indexed Nested-Loop (INL) Algorithm

Figure 4 contains the pseudo-code for the INL algorithm, which is a variant of a nested loop algorithm. Each occurrence of the first word w_1 in a context interval is a partial witness. For each such word, INL attempts to construct a complete witness by adding a contiguous sequence of ignored markup and other phrase words in order. In particular, for each context interval, we probe L_{w_1} to find the first word in the phrase contained in the context interval (lines 1–3) and construct a partial witness containing this word (lines 6–7). We then probe the index containing the ignored markup (L_M) and the index containing the next word in the phrase $(\mathrm{L}_{w_{(matchPos+1)}})$, attempting to extend the current witness contiguously. If we cannot extend the witness, we discard it and start again (lines 11–12). We continue extending the witness until every word is matched, then add the complete witness to the context interval's set of witnesses (line 17–19). When no more witnesses can be matched in the current context interval, we output the context interval and its set of witnesses (line 21) and continue with the next context interval (line 1).

Note that the outer-loop of the INL algorithm is evaluated once for each context interval and that each witness is constructed independently of all other witnesses. This may result in redundant work, for example, when a context or annotation interval is nested within another context interval (as in Figure 2), because the intervals of the nested witness are traversed

```
{(1,44),
  { [ 6, 7, 8, 9, 10, 11,
       (12, 38), 39, 40, 41, 42 ],
     [ 16, 17, 18, 19, 20, 21,
       22, 23, 24, 25 ] } }
```

(a)

L`SPEECH`	L`LINE`	L`COMMENT`	L`to`	L`be`	...	$L_{question}$
(1,44)	(5,43)	(12,38)	(6,6)	(7,7)		(25,25)
			(10, 10)	(11,11)		(42,42)
			(16, 16)	(17,17)		
			(20, 20)	(21,21)		

(b)

Figure 3: (a) Example output of phrase matching algorithms, (b) Indices for document fragment in Figure 2

```
for each context interval $i_c$ in $L_C$ {
  witnessSet = { }
  index probe $Lw_1$ to find first interval $i_1$
     such that descendant($i_1$, $i_c$)
  repeat {
    matchPos = 1;
    $m$ = [ $i_1$ ];
    repeat {
      probe ($Lw_{(matchPos+1)}$ ∪ $L_M$) to find
        $i_2$ with $i_2$.start = last($m$).end+1;
      if (no match found) break;
      if ($i_2 \in Lw_{(matchPos+1)}$) matchPos++;
      $m$ = append($m$, $i_2$);
      /* Matched last word in phrase */
    } until (matchPos = $q$)
    /* If a complete witness is found, save it */
    if (matchPos = $q$)
      witnessSet = witnessSet ∪ { $m$ };
    $i_1$ = next($Lw_1$)
  } until not(descendant($i_1$, $i_c$))
  output ($i_c$, witnessSet)
}

descendant($i_1$, $i_2$) {
  $i_1$.start > $i_2$.start and $i_1$.end < $i_2$.end
}
```

Figure 4: Indexed nested loop algorithm

once when matching the witness itself and one or more times when matching the witness in which it is nested.

Indexed nested loop algorithms are well studied and understood for relational databases. The INL algorithm is expected to have similar characteristics when the XML data is akin to relational data (e.g., there is no nesting of contexts) and when there are few tags and annotations to ignore. In cases where XML's heterogeneity is instantiated, the INL algorithm tends to perform a large number of probes, many of which may be redundant. We study this behavior experimentally in Section 5.

4.2 PIX Stack-Based Merge Algorithm

Just as the INL algorithm is analogous to index-nested loop algorithms for relational data, the PIX stack-based merge algorithm is analogous to traditional sort-merge join algorithms. Like all sort-merge algorithms, the PIX algorithm scans its input only once. In particular, PIX scans L, the combined list of words and ignored markup in order and uses a stack S to keep track of nested context and annotation intervals and partial witnesses as they are identified within the nested intervals. These structures are defined as:

L: Priority queue over $\cup(L_C, (\cup_j L_{w_j}), L_M)$.

S: Stack of (`interval`, `witnessSet`, `matchSet`)s.

The list L is implemented as a priority queue over L_C, L_M, and L_{w_j}. Each entry on the stack S is an (`interval`, `witnessSet`, `matchSet`) tuple, where `interval` is a context or annotation interval i, `witnessSet` is a set of the complete witnesses matched in i, and `matchSet` is a set of matches $\{m\}$. A match m is a (`partialWitness`, `matchPos`) pair, where `partialWitness` is an interval list and `matchPos` is the index of the last phrase word matched in the partial witness. Because the first word in a phrase may be repeated within the phrase, we maintain a *set* of partial witnesses. For example, given the phrase "w_1 w_2 w_1 w_3" and the input "w_1 w_2 w_1' w_2' w_1'' w_3", both $[w_1\, w_2\, w_1']$ and $[w_1']$ are valid partial witnesses. We refer to the interval in the top entry of the stack as the "top interval" and, similarly, for the "top witness set" and "top match set".

Figure 5(left-top) gives the PIX algorithm. The PIX algorithm scans L, the combined list of words and ignored markup in order (lines 1–2). The interval i is either a new context interval (lines 3–7) or a word or ignored markup (lines 8–20).

If i is a context interval and i is not a descendant of the top interval, then the top interval and its partial witnesses will never be complete, so we clean the stacks by calling the procedure `output-and-clean` (lines 4–6), which pops S until i is a descendant of the top interval or S is empty (lines 25–33). As context intervals are popped from S, their witness sets are output (lines 27–28) and are propagated up the stack to their closest containing interval (lines 30–31). After cleaning the stack, we create a new interval in which to match phrases by calling `new-interval` on line 7.

If i is either a phrase word or ignored markup and S is empty, we discard the interval, because there is no current context (line 9). Otherwise, if i is not a descendant of the top interval, we again clean the stacks (lines 10–11).

Once we have a word or markup i that is a descendant of the top interval, we attempt to create or extend a partial witness. If i is markup, we call `extend-with-markup` (lines 13–14). In `extend-with-markup`, we attempt to extend each partial witness in the top match set (lines 43–45). If some

```
1. while (not(empty($L$))) {
2.   $i$ = remove-first($L$);
3.   if ($i \in L_C$) { /* $i$ is context interval */
4.     if (not(empty($S$)) &&
5.         not(descendant($i$,top($S$).interval)))
6.       output-and-clean($i$);
7.     new-interval($i$);
8.   } else { /* $i$ is word or ignored markup */
9.      if (empty($S$)) break;
10.     if (not(descendant($i$, top($S$).interval)))
11.       output-and-clean($i$);
12.     /* $i$ is descendant of top($S$).interval */
13.     if ($i \in L_M$) {
14.       extend-with-markup($i$);
15.       if ($i \in L_{a\_j}$)
16.         /* $i$ is nested annotation */
17.         new-interval($i$);
18.     } else if ($i \in L_{w\_pos}$)
19.       extend-with-word($i$, $pos$)
20.   }
21. }
22. if (not(empty($S$))) output-and-clean((0,0));
23.
24. output-and-clean($i$) {
25.  repeat {
26.    $c$ = pop($S$);
27.    if ($c$.interval $\in L_C$) /* context interval */
28.      output($c$.interval,$c$.witnessSet);
29.    /* Propagate nested witnesses up stack */
30.    top($S$).witnessSet =
31.      top($S$).witnessSet $\cup$ $c$.witnessSet;
32.  } until (empty($S$) or
33.           descendant($i$,top($S$).interval));
34. }

35. new-interval($i$) {
36.   push(($i$, {}, {}), $S$);
37. }

38. discard-partial-match($m$) {
39.  top($S$).matchSet = top($S$).matchSet - {$m$}
40. }

41. extend-with-markup($i$) {
42.  for each $m \in$ top($S$).matchSet {
43.    if ($i$.start =
44.        last($m$.partialWitness).end + 1)
45.      $m$.partialWitness =
           append($m$.partialWitness, $i$);
46.    else
47.      discard-partial-match($m$)
48.  }
49. }

50. extend-with-word($i$, $pos$) {
51.  if ($pos$ = 1) {
52.    top($S$).matchSet =
         top($S$).matchSet $\cup$ ([ $i$ ], 1);
53.  } else {
54.    for each $m \in$ top($S$).matchSet {
55.      if ($m$.matchPos + 1 = pos and $i$.start =
56.          last($m$.partialWitness).end + 1) {
57.        $m$.partialWitness =
             append($m$.partialWitness, $i$);
58.        $m$.matchPos++;
           /* Once matched complete phrase */
59.        if ($m$.matchPos = $q$) {
60.          /* Add to top witness set */
61.          top($S$).witnessSet = top($S$).witnessSet
               $\cup$ { $m$.partialWitness }
62.          discard-partial-match($m$)
63.        }
64.      } else
65.        discard-partial-match($m$)
66.    }
67.  }
68. }
```

Figure 5: PIX stack-based merge algorithm and auxiliary functions

partial witness cannot be extended, it is discarded (line 47). An ignored annotation, in addition to extending a partial witness, may contain witnesses itself, so we push a new interval for the annotation (lines 15–17) and continue matching phrases within the annotation. Phrase matching within an annotation interval is identical to that within a context interval, except that witnesses within an annotation are propagated up the stack and output along with all the other witnesses in the nearest context interval.

If i is a word, we attempt to create or extend a partial witness by calling `extend-with-word` (lines 18–19). If i denotes the first word w_1, `extend-with-word` starts a new partial witness (lines 51–52), otherwise, it attempts to extend contiguously each partial witness (lines 53–58). If a witness is completed, it is added to the witness set of the top interval (lines 59–63) If some partial witness cannot be extended, it is discarded (lines 64-66). When L is exhausted, we output the remaining complete witnesses on the stack (line 22).

The PIX algorithm generalizes the structural join algorithms of [1] by taking the order of words in a phrase into account. All the algorithms use stacks to identify ancestor-descendant pairs by sequentially scanning through interval lists, but only PIXbuilds sets of partial witnesses and incrementally extends them.

4.3 Analysis of PIX Algorithm

The PIX algorithm traverses once each of the interval lists of phrase words, ignored tags, ignored annotations, and contexts. It maintains in memory one stack, whose maximum depth is bounded by the maximum nesting depth of context and annotation intervals. Thus, the stack is bounded by the nesting depth of the XML document. Each entry on the stack maintains a set of partial witnesses, consisting of one or more matches of the phrase words and any ignored markup. The number of partial witnesses is bounded by the number of occurrences of the first word in the

```
 1. extend-with-word(i, pos) {
 2.  for each m ∈ top(S).matchSet {
 3.    if (m.matchPos + 1 = pos and i.start =
 4.          last(m.partialWitness).end + 1) {
 5.       m.partialWitness =
             append(m.partialWitness, i);
 6.       m.matchPos++;
          /* Once matched complete phrase */
 7.       if (m.matchPos = q) {
 8.         /* Add to top witness set */
 9.         top(S).witnessSet = top(S).witnessSet
                 ∪ { m.partialWitness }
10.         discard-partial-match(m)
11.       }
12.    } else if (m.skipped + i.start -
             last(m.partialWitness).end - 1 <= k) {
13.       m.skipped += i.start -
              last(m.partialWitness).end - 1;
14.       m.partialWitness =
              append(m.partialWitness, i);
15.    } else {
16.       discard-partial-match(m)
17.    }
18.    if (pos = 1) {
19.       top(S).matchSet =
20.          top(S).matchSet ∪ ([ i ], 1, 0);
21.    }
22.  }
23. }
```

Figure 6: Procedure `extend-with-word` modified to support proximity matching

phrase. The size of each partial witness depends on the number of words in the phrase, and the number of occurrences of intervening markup to be ignored. When this is small, which is often the case, the stacks fit in main memory. The I/O complexity of the PIX algorithm is, hence, linear in the sum of the input and output sizes. This makes it optimal among all algorithms that read their entire input and produce the complete output.

4.4 PIX with Proximity Phrase Matching

We can easily extend the PIX algorithm to support phrase matching within a proximity of k words. We include a counter (`skipped`) in each match m in `matchSet`; the counter contains the number of words that have been skipped while constructing the m's partial witness. A partial witness can be extended as long as its `skipped` value is $\leq k$.

Figure 6 contains the procedure `extend-with-word` modified to support word proximity. We attempt to extend contiguously each partial witness just as in the original procedure (lines 3–12). If the partial witness cannot be contiguously extended with the new word, but the number of skipped words would not exceed k, we extend the partial witness and increment the number of skipped words (lines 12–15). Otherwise, the partial witness is discarded, because it cannot be extended and its proximity limit is exceeded (line 16). Finally, if i denotes the first word w_1, we start a new partial witness (lines 18–21) – we do this after examining the other partial witnesses as the first word might also extend some of these as a skipped word.

Consider the data "w_1 w_2 w'_1 w_3 w'_2 w'_3 w_4" and the query phrase "w_1 w_2 w_3 w_4" matched within three words. After the word w'_3 is processed, there are two partial witnesses: $([w_1, w_2, w'_1, w_3, w'_2, w'_3], 3, 3)$ and $([w'_1, w_3, w'_2, w'_3], 3, 1)$. In the first partial witness, the words w'_1, w'_2, w'_3 are skipped words; in the second partial witness, the word w_3 is a skipped word. Each of these partial witnesses can be extended with w_4 to obtain complete witnesses. Note that this algorithm reports the *first* witness beginning with a particular occurrence of w_1, but does not report all overlapping witnesses. For example, it does not report w_1 w_2 w'_1 w_3 w'_2 w'_3 w_4 in which the phrase words w_2, w'_1, w_3 are the skipped words.

4.5 Ranking Results

An important aspect of keyword and phrase matching in IR is to associate a score with each match. In principle, PIX can apply any user-defined ranking function during matching. For concreteness, we briefly outline one possible strategy for ranking witnesses that accounts for the relative importance of phrase words, the importance and size of ignored markup, and the number of skipped words. Our strategy assigns IDF weights to individual phrase words, where the IDF weight is the inverse of the number of context nodes containing occurrences of the phrase word in the corpus. In a witness, a phrase word is assigned a TF-IDF weight identical to its IDF weight (since there is just one occurrence). Each ignored markup is assigned IDF weights similarly, by counting the number of context nodes containing distinct occurrences of the markup in the corpus. In a witness, the TF-IDF weight associated with an ignored markup is the product of (i) the number of occurrences of the markup in the witness and (ii) its IDF weight. Finally, the score of a witness is computed as the difference of two terms: (i) the sum of the TF-IDF weights of all the query phrase words, and (ii) the sum of the TF-IDF weights of all the ignored markup. This strategy assigns a lower score to witnesses that ignore more important markup (as measured by its IDF weight) and more occurrences of markup. In proximity phrase matching, the score of a witness is inversely proportional to the number of skipped words.

5 Experiments

We compare the INL and PIX algorithms experimentally to understand their relative performance and the key parameters influencing it. When used to implement relational joins, index-nested loop tends to out-

File	Doc Size	Index Size	Context Node	#Context Nodes	Phrase	# Witnesses	#Ignored Tags	#Ignored Annots
WSJprd	1.5MB	2.3MB	<NP>	13	"country funds"	13	0	0
WSJmrg	2.5MB	4.5MB	<FILE>	1	"Pierre Vinken ..."	1	9	2
Brown	15.0MB	25.7MB	<PP>	8	"in the United States"	8	2	0

Table 2: Characteristics of Treebank Data and Queries

perform sort merge when the outer-most relation is small and the inner relations are accessible via indices, and sort merge tends to outperform index-nested loop when the input relations are sorted and are of comparable size. When used to implement phrase matching, we expect the INL and PIX algorithms to behave similarly. First, we present experiments that support these expectations. Second, we consider how varying parameters, that are unique to XML and our customized phrase matching, impact the relative performance of the INL and PIX algorithms.

5.1 Platform and Data

Both the INL and PIX algorithms are implemented in Java. We use the Berkeley DB package [6] to build and access the indices. Berkeley DB is implemented in C and a Java interface is provided by wrapping the C functions using the Java native interface. All the experiments were executed on a Dell computer with a 2Ghz CPU, 1GB of memory, and a 34GB drive running Windows 2000. We executed each query five times with a cold cache and computed the average execution time. Times are in seconds and sizes in megabytes.

Our data sets include real data from the Treebank corpus published by the Linguistic Data Corporation [18] and synthetic data generated by the XMach XML document generator [8]. The Treebank data is the result of natural-language analysis of text excerpts from the Wall Street Journal and from the Brown corpus of popular literature. Table 2 describes the documents. The "WSJprd" and "Brown" documents contain the corresponding texts labeled with syntactic annotations. The "WSJmrg" document is the same as "WSJprd" additionally labeled with parts-of-speech annotations (i.e., "WSJmrg" has more markup).

Synthetic data is generated by the XMach XML document generator. The generator takes as input a list of words, their frequencies in the generated document, and an output document size, and generates a document of the given size containing text with the given word frequencies. The baseline query (Q_B) matches the phrase "w_1 w_2", ignores one tag (tag), and ignores one annotation (annot). To generate our baseline documents, we replace the most frequently occurring word in the XMach-generated document with a witness of the form: <tag>w_1</tag><annot>...</annot>w_2.

The small baseline document (D_{small}) is 150KB in size and contains one context node (the immediate child of the root node) and 680 witnesses (because the most frequent word occurs 680 times); the second baseline document is 30MB (D_{big}). Neither document contains nested context nodes or annotation nodes nested at a depth greater than one. All other documents used in the experiments are variants of these two baseline documents, in which other words are replaced with the witness described above, tags or annotations to ignore, or context nodes. We specify in the following sections how other documents are constructed from these baseline documents.

	Witnesses reported	
	One	All
INL	0.06s	0.06s
PIX	25.21s	27.10s

Table 3: Query time for WSJmrg data set

We report the time to find all witnesses in all context nodes (all-wits). In some cases, we report the time to find just the first witness in all context nodes (one-wit), but in general, the times to report the first witness are similar to reporting all witnesses and therefore do not differentiate the algorithms.

5.2 Applicability of Known Results

We expect the INL algorithm to outperform the PIX algorithm when the inverted index of the first phrase word is small, because the INL algorithm will perform few probes of the other indices. In the WSJmrg data set, the phrase query is "Pierre Vinken will join the board", and the context is those nodes with tag NP. There are nine distinct tags to ignore and two distinct annotations. In this data set, the first word "Pierre" occurs exactly once in the document, meaning that the INL algorithm does a minimal number of probes, whereas the PIX algorithm scans the combined list of words and ignored markup to find the first occurrence of the first word. The resulting query times (in Table 3) differ significantly.

We confirmed this difference on synthetic documents varying in size from 2-20MB in which the lengths of the inverted indices L_{w_1} and L_{w_2} varied. When L_{w_1} is substantially smaller than L_{w_2}, the INL algorithm outperforms the PIX algorithm, because INL makes only a few probes of L_{w_2}, whereas the PIX algorithm sequentially reads the entire lists. As the length of L_{w_1} approaches the length of L_{w_2}, the PIX algorithm outperforms INL.

One way to improve the performance of the PIX algorithm is to build a skip list for each word and ig-

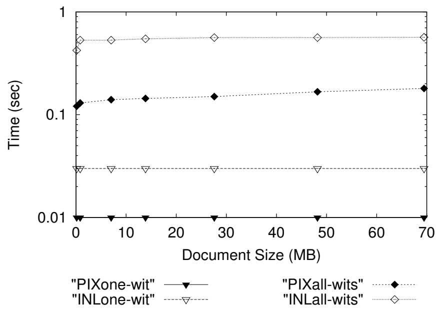


Figure 7: Increasing document size

nored tag over the priority queue L. This effectively combines the priority queue L with the individual lists L_C, L_M, and L_{w_j} and would permit the PIX algorithm to skip over long sequences of words or markup at the head of the priority queue L that cannot contribute to a witness. We plan to do this in future work on PIX and expect this will substantially improve PIX when the relative lengths of word lists differ significantly.

As expected, both algorithms are insensitive to changes in document size when the number of context nodes and the number of witnesses per context node remain constant. In Figure 7, we create documents from 150KB to 100MB in size by appending to D_{small} copies of a second document that has no words in common with D_{small}. As in D_{small}, each resulting document contains one context node with 680 witnesses. Because both algorithms use indices, they retrieve only the relevant words and markup to match a phrase.

5.3 Exploring Variability in Data and Queries

Next, we explore the impact of various parameters of XML documents and phrase queries on the relative performance of the two algorithms.

Witnesses and context nodes

First, we consider the effect of the number of context nodes and number of witnesses in the data. Recall that our baseline query matches the phrase "$w_1 w_2$" and ignores the tag (`tag`) and the annotation (`annot`).

In the first test, we start with the baseline document D_{big} and fix the number of context nodes to one and vary the number of witnesses, by replacing the second (third, fourth, etc.) most frequent word in D_{big} with the witness `<tag>`w_1`</tag><annot>...</annot>`w_2. The resulting documents contain 204000 to 530255 witnesses. In the second test, we start with the baseline document D_{small} and fix the number of witnesses per context node to 680. We increase the number of context nodes from 1 to 121 by replicating D_{small} to create new documents. Figure 8 plots the results of evaluating Q_B on these documents. Both PIX and INL grow linearly with increasing numbers of context nodes and witnesses per context node, but PIX is about four times faster than INL and this difference increases with document size. This is due to the cost of index probing by INL, but also due to the increase in the number of context nodes, because INL probes every list for each context node.

Correl- ation	Replace	with	#Context Nodes	#Witnesses/ Context Node
(0,0)	b_1 & b_2	w_1	1–200	63–19000
	b_3 & b_4	w_2		
(1,1)	b_1 & b_2	$w_1 w_2$	1–350	680–238000
(1,0.5)	b_1	$w_1 w_2$	1–250	1046–260000
	b_2	w_2		
(0.5,1)	b_1	$w_1 w_2$	1–250	1046–260000
	b_2	w_1		

Table 4: Documents with varying word correlations

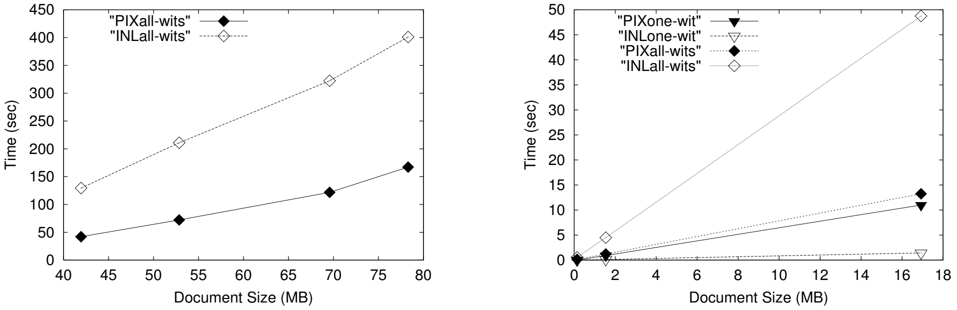
Next, we start with the baseline document D_{small} and generate documents that vary the correlations between the two words w_1 and w_2 in the match phrase. We generate a new document by replacing the four words b_1, b_2, b_3 and b_4 in D_{small} that occur with the highest frequencies with the match phrase "$w_1 w_2$" or the individual words w_1 or w_2. Then we replicate these four documents to create documents up to 60MB in size. Table 4 describes the generated documents; the correlation specifies the ratio of the number of co-occurrences of w_1 and w_2 to the number of occurrences of each word, i.e., $\#(w_1\ w_2)/\#w_1$ and $\#(w_1\ w_2)/\#w_2$.

Figure 9 plots the results of evaluating Q_B on these generated documents. We observe that PIX is up to two times faster than INL. We note that in the (1,0.5) test, the first word list (L_{w_1}) is half as long as the second word list (L_{w_2}), and on the largest document, PIX is only slightly faster than INL. This example complements the results in Section 5.2 in which we observed that INL does well when the first word list is much shorter than all other lists.

Finally, we fixed the number of context nodes and witnesses per context node, and varied the number of words in the phrase query from two to seven. We observed that PIX is about four times faster than INL and this difference increased with the number of phrase words, because INL probes all the word lists to construct a complete witness. (Graph omitted due to space constraints).

Nesting of annotations and context nodes

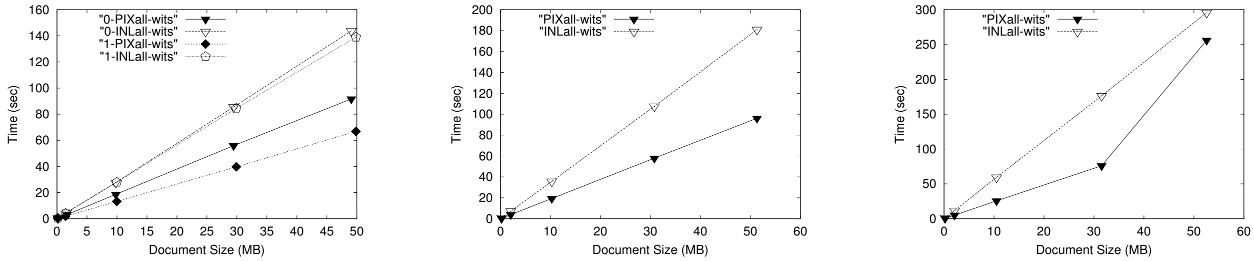
Next, we consider the effect of varying the nesting of annotations and context nodes. We begin with the baseline document D_{big}, and generate three new documents. In the first, we replace each occurrence of `<annot>...</annot>` by `<annot><tag>`w_1`</tag><annot>... </annot>`w_2`</annot>`. This creates another level of annotation nesting as well as a new witness. In the second and third documents, we again double



(a)

(b)

Figure 8: (a) Varying number of witnesses and (b) varying number of context nodes



(a) (0, 0) and (1, 1)

(b) (0.5, 1)

(c) (1, 0.5)

Figure 9: Varying word correlation

the nesting levels, thus we have `<annot>` elements at nesting depths one, two, four, and eight, with a witness at each level. Figure 10(a) plots the results of evaluating Q_B on these documents.

Lastly, we vary the nesting of context nodes. We start with the baseline document D_{big}, fix the total number of context nodes, but increase the nesting depth of context nodes from 1 to 2, 4, and 8. Figure 10(b) contains the query times for Q_B. In the "low" test, we set the number of witnesses per context node to 136000, and in the "high" test, we set the number of witnesses to 236400. In both cases, as the nesting level of context nodes increases, INL increases linearly, whereas PIX remains constant. For a witness contained in a context node nested n levels deep, the INL algorithm accesses the witness n times – once for each nested context node – wheres the PIX algorithm accesses the witness once and passes the nested witness up the stack to ancestor context nodes.

Experiments on the real data support this observation. In the WSJprd data set, most of the `<NP>` context nodes are nested, and all the witnesses are matched in context nodes at nesting depth three. The match phrase is "country funds" and there are no tags or annotations to ignore and thirteen context nodes each containing one witness. Table 5 gives the query times for INL and PIX. PIX is more than four times faster than INL, because INL is constructing each witness three times as often as PIX.

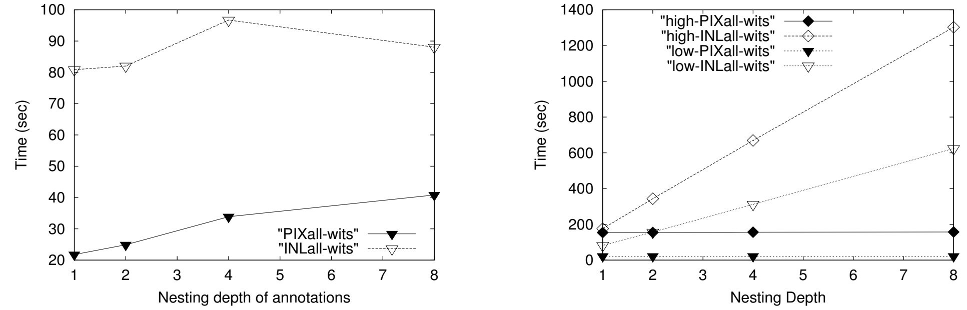
		Witnesses reported	
		One	All
WSJprd	INL	3.91s	4.03s
	PIX	0.45s	0.95s
Brown	INL	16.12	16.12
	PIX	15.12	15.13

Table 5: Query time for WSJprd and Brown data sets

In the Brown data set, the `<PP>` context nodes are never nested. The phrase to match is "in the United States" and there are two distinct tags to ignore and no annotations to ignore. In this data set, there are eight context nodes containing one witness each, and all such context nodes are at nesting level one. In Table 5, we can see that PIX is only slightly faster than INL as INL does little redundant work.

6 Discussion

We have presented a customizable approach to phrase matching, tailored to XML documents, that permits specification of the phrase to be matched exactly or within a proximity of k words, the document context in which to restrict the phrase match, and the markup (tags and annotations) that should be ignored within the context. To support such XML phrase matching,



(a) Nested annotations in data

(b) Nested context nodes in data

Figure 10: Varying nesting in data

we have designed an efficient stack-based merge algorithm, PIX, that utilizes inverted indices on phrase words and tags. We have demonstrated the superiority of the PIX algorithm over the competing INL algorithm both analytically and experimentally. An interesting observation is that, since our PIX algorithm traverses the input only once, it can be adapted to streaming processing of XML documents.

There are many interesting directions of future work on the topic of combining document-style and structured querying in XML. We discussed briefly the issue of ranking XML phrase matches in the presence of ignored markup and proximity matching. An interesting problem is to adapt the PIX algorithm for efficiently computing top-K matches, under a ranking metric. This would require early pruning of context nodes and partial witnesses that are guaranteed not to contribute to the final answer. One can also extend PIX to permit a more sophisticated specification of the context nodes and the markup to be ignored, using XPath and XQuery expressions. With this increased expressive power, efficient query evaluation becomes even more challenging. Another open problem is identifying an effective way to combine proximity phrase matching with proposals for approximate matching of structured XML queries (see, e.g., [3]).

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] S. Amer-Yahia and P. Case. XQuery and XPath full-text use cases. W3C Working Draft. Available from http://www.w3.org/TR/xmlquery-full-text-use-cases, Feb. 2003.

[3] S. Amer-Yahia, S. Cho, and D. Srivastava. Tree pattern relaxation. In *EDBT*, 2002.

[4] R. Baeza-Yates and G. Navarro. XQL and proximal nodes. In *SIGIR Workshop on XML and Information Retrieval*, 2000.

[5] R. Baeza-Yates and G. Navarro. Integrating content and structure in text retrieval. *SIGMOD Record*, March 1996.

[6] The Berkeley DB system. Distributed by Sleepcat Software. http://www.sleepycat.com/.

[7] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. W3C Working Draft. Available from http://www.w3.org/TR/xquery, May 2003.

[8] T. Böhme and E. Rahm. XMach-1: A benchmark for XML data management. In *Proc. of German database conference BTW*, 2001. http://dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html.

[9] J. Bosak. The plays of Shakespeare in XML. http://www.oasis-open.org/cover/bosakShakespeare200.html.

[10] J.-M. Bremer and M. Gertz. XQuery/IR: Integrating XML document and data retrieval, In *WebDB*, 2002.

[11] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *SIGIR*, 1995.

[12] D. Florescu, D. Kossman, and I. Manolescu. Integrating keyword search into XML query processing. In *WWW*, 1999.

[13] N. Fuhr and K. Grossjohann. XIRQL: An extension of XQL for information retrieval. In *SIGIR*, 2001.

[14] Galax. An implementation of XQuery. http://db.bell-labs.com/galax/.

[15] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[16] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, 2003.

[17] The Library of Congress. XML use cases. http://www.loc.gov/crsinfo/xml/lc_usecases.html.

[18] M. Marcus, et al. Treebank-2, LDC catalog no.: LDC95T7. CD-ROM. Philadelphia: Linguistic Data Consortium, 1999.

[19] G. Salton and M. J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, New York, 1983.

[20] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *EDBT*, 2002.